关联规则分析与 Apriori 算法

CS245　数据科学基础　陆朝俊

1　问题描述

关联规则分析是数据挖掘中活跃的研究方法之一，其目的是在一个数据集中找出各项之间的关联关系（这种关系一般没有在数据中直接表示出来）。Apriori 是关联规则分析中最常用也是最经典的挖掘频繁项集的算法，在本次作业中，我将实现 Apriori 算法，从交易数据集中发现频繁项集，并生成相应的关联规则。

2　解决方案[1]

2.1　数据集的准备

为验证 Apriori 算法实现的正确性，我根据一定的规则生成了一个模拟交易数据集，数据集的每个实例代表一条交易记录，共 100 条记录（在图 2 对应的实验中会改变数据集大小）。我将商品分为食品（bread、milk、apple、orange、beer），电器（TV、PC、phone、fridge、ele_oven）和工具（scissors、stapler、plate、knife、glue）三类，各类商品按照一定的出现及组合概率生成相应的交易记录，具体的参数设定可参见附录 A.1。

我认为对于交易情况的分析，仅使用模拟生成的数据集难以取得真实的结果，因此我额外在 Groceries 数据集上执行了 Apriori 算法。Groceries 数据集是内置于 R 语言的关联分析数据集，来源于某杂货店一个月的真实交易记录，包含 9835 条交易记录及 169 种商品。我将其从 R 语言包中提取出并重组为.csv 格式，再使用 Apriori 算法进行分析。

2.2　Apriori 算法

2.2.1　算法描述

Apriori 算法是最经典的挖掘频繁项集的算法，实现了在大数据集上可行的频繁项集挖掘和关联规则提取，其核心思想是**通过连接产生候选项与其支持度，然后通过剪枝生成频繁项集**，步骤主要为：

1. 通过自连接获取候选项集，第一次迭代的候选项集即为数据集中的项，之后迭代时的候选项集由前次迭代的频繁项集自连接得出；
2. 每次迭代时的频繁项集通过候选项集剪枝得出，对于候选项集的每条记录，若它或它的某个子集的支持度小于设定的阈值，则被删除；
3. 由频繁项集产生强关联规则（经过上个步骤后满足给定的置信度阈值的关联规则）。

2.2.2　Python 实现细节

在这一小节中，我将结合我的部分 Python 代码对 Apriori 算法的实现进行介绍。

首先，我将所有频繁 1-项集组合起来，称为 L_1 项集，并存储为 Python 中的集合类型（frozenset 类型用于防止误操作更改其中的值）：

[1]本次作业的主要代码实现可参见附录 A.2

```python
def _init_data(self):
    deal_list = list()
    item_set = set()

    for d in self.data:
        deal = frozenset(d)
        deal_list.append(deal)

        # 产生 L1 项集
        for item in deal:
            item_set.add(frozenset([item]))

    return item_set, deal_list
```

得到 L_1 项集后，我开始迭代求解 $L_2...L_k$ 项集。第 i 次迭代时，我对 L_i 进行自连接操作，求出 i+1 项集的集合 C_{i+1}，自连接的规则为：设 s_1 和 s_2 是 L_i 中的两个频繁 i 项集，且前 i-1 个项相同，仅第 i 项不同，连接 s_1 和 s_2 得到一个 i+1 项集 C_{i+1}。Python 中通过循环使用并集操作并判断集合大小可以简洁地实现自连接过程：

```python
set([i.union(j) for i in item_set for j in item_set if len(i.union(j)) == id])
```

自连接构造出 C_{i+1} 后，我通过扫描交易数据集检测出 C_{i+1} 中的频繁 i+1 项集，得到 L_{i+1}，此处存在一个优化点：若有某元素的子集不在 L_i 中即可删除。

```python
# 根据支持度阈值删除项集
    def _remove_item_set(self, item_set, deals, freq_set):
        res = set()
        local_set = defaultdict(int)

        # 统计
        for item in item_set:
            for d in deals:
                if item.issubset(d):
                    freq_set[item] += 1
                    local_set[item] += 1

        # 删除
        for item, count in local_set.items():
            support = float(count) / len(deals)
            if support >= self.sup:
                res.add(item)

        return res
```

将上述迭代过程不断重复，直到项集大小达到商品个数，迭代结束，得到最终结果。

为验证 Apriori 算法实现的正确性，我先使用蛮力算法在模拟交易数据集上运行一次，将结果与 Apriori 算法的结果进行比较。验证算法的正确性后，在 Groceries 数据集上我则直接使用 Apriori 算法进行关联规则分析。

3 实验及结果

3.1 模拟数据集

我将支持度和置信度的阈值分别设置为 0.1 及 0.6，蛮力算法的运行结果为：

表 1: 蛮力算法在模拟交易数据集下发现的频繁项集

频繁项集	支持度	频繁项集	支持度
scissors	0.25	apple	0.18
beer	0.22	fridge	0.25
ele_oven	0.26	stapler	0.30
phone	0.22	glue	0.23
plate	0.23	scissors, knife	0.15
PC	0.22	ele_oven, TV	0.17
TV	0.29	scissors, stapler	0.15
bread	0.19	glue, stapler	0.16
orange	0.21	stapler, plate	0.15
knife	0.27	stapler, knife	0.15
milk	0.18		

表 2: 蛮力算法在模拟交易数据集下发现的关联规则

关联规则	置信度
glue –> stapler	0.70
scissors –> knife	0.60
ele_oven –> TV	0.65
plate –> stapler	0.65
scissors –> stapler	0.60

得到以上的参考结果后，我使用自己实现的 Apriori 算法和同样的参数（支持度 0.1，置信度 0.6）对模拟交易数据集进行分析，所得结果见表 4 和表 3。

表 3: Apriori 算法在模拟交易数据集下发现的关联规则（按置信度排序）

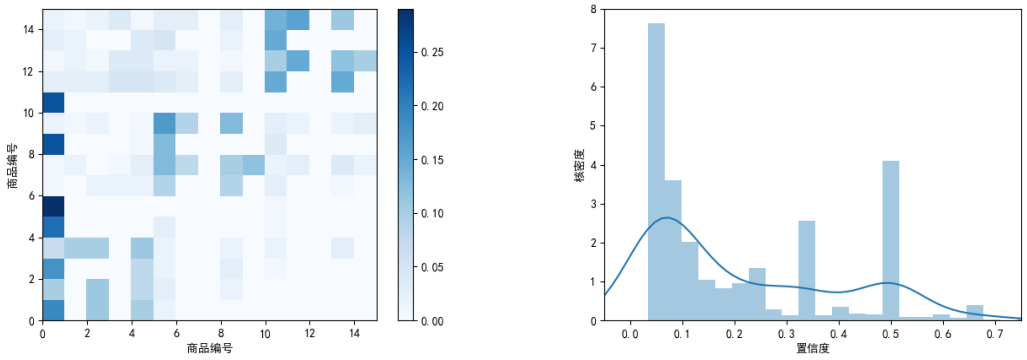
关联规则	置信度
scissors –> stapler	0.60
scissors –> knife	0.60
plate –> stapler	0.65
ele_oven –> TV	0.65
glue –> stapler	0.70

表 4: Apriori 算法在模拟交易数据集下发现的频繁项集（按支持度排序）

频繁项集	支持度	频繁项集	支持度
stapler, scissors	0.15	PC	0.22
knife, scissors	0.15	phone	0.22
stapler, knife	0.15	glue	0.23
plate, stapler	0.15	plate	0.23
stapler, glue	0.16	scissors	0.25
ele_oven, TV	0.17	fridge	0.25
milk	0.18	ele_oven	0.26
apple	0.18	knife	0.27
bread	0.19	TV	0.29
orange	0.21	stapler	0.30
beer	0.22		

将表 1，2的结果排序后与上表对比，容易发现二者一致。考虑到模拟交易数据集在生成时的随机性，这个结果可以证明我实现的 Apriori 算法的正确性。

为探索该数据集中所有项集和相应关联规则的分布情况，我将支持度和置信度的阈值设为 0，运行 Apriori 算法，并作出项集分布的热度图以及关联规则置信度的核密度图，结果可参见图 1（热度图中编号从 1 至 15 的商品分别为 bread、milk、apple、orange、beer、TV、PC、phone、fridge、ele_oven、scissors、stapler、plate、knife、glue；作核密度图时我采用高斯核）。



(a) 项集　　(b) 关联规则

图 1: 模拟交易数据集中项集和相应关联规则的分布（由于之前计算出的频繁项集最多包含两项商品，因此此处只作出二维热度图）

从图 1(a)中容易看出，支持度相对高的项集集中于编号为 1，6，11 的商品处，分别为（bread，TV，scissors），并且支持度高的点集中于对角线附近，表明仅有一个元素的项集有着更高的支持度。图 1(b)反映出关联规则对应的置信度普遍偏低（集中于 0-0.4，较大部分在 0.1 以下），虽有部分达到 0.5 的置信度，但置信度稍高（0.6 以上）的关联规则仅有表 3中所列的数个。

根据理论分析，Apriori 算法相对于蛮力算法有着极大的效率优势，为直观地探索这种优势，我在不同的交易记录数下运行两种算法进行频繁项集挖掘，对比消耗的时间，结果参见图 2。

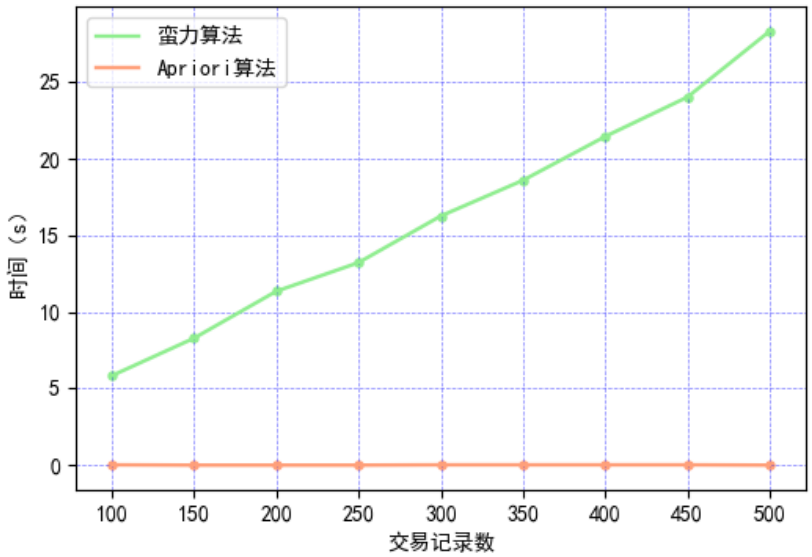


图 2: Apriori 算法与蛮力算法进行频繁项集挖掘效率对比

从图中可以看出，在同样的数据规模下，Apriori 算法消耗的时间远少于蛮力算法，上图的比较仅改变了数据集的交易记录数量，若增加总交易货品种类，蛮力算法的复杂度将呈指数型增长，Apriori 算法的效率优势会更加明显（限于计算资源，本次作业中暂不进行不同商品类数下的效率比较）。

3.2 Groceries 数据集

在模拟交易数据集上确认 Apriori 算法实现的正确性后，我将算法应用到真实数据上。考虑到 Groceries 数据集中商品种类相对于实例较多的特点，即每种商品出现的次数占总交易记录数的比例可能偏低，我选择了较小的支持度（0.05），置信度选择为 0.2。运行结果参见表 6 和表 5。

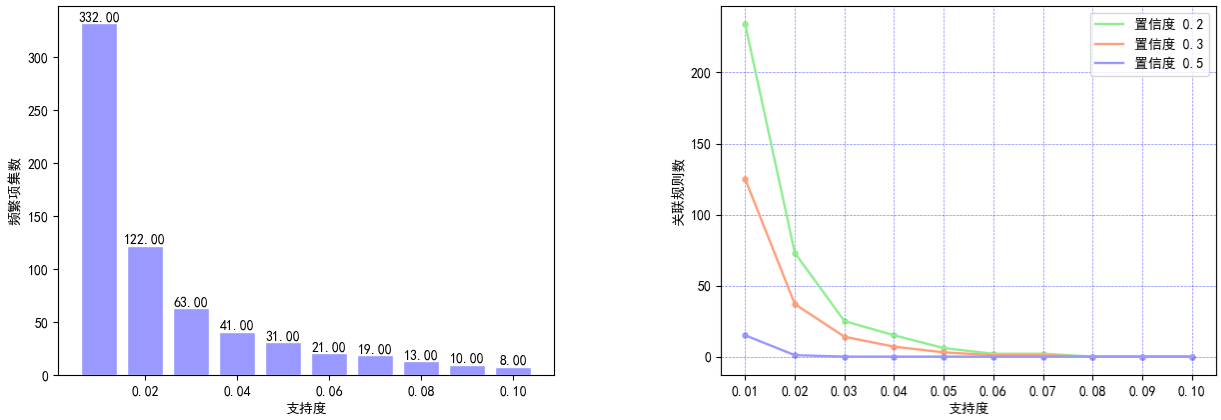
表 5: Apriori 算法在 Groceries 数据集下发现的关联规则（按置信度排序）

关联规则	置信度
whole milk –> yogurt	0.22
whole milk –> rolls/buns	0.22
whole milk –> other vegetables	0.29
rolls/buns –> whole milk	0.31
other vegetables –> whole milk	0.39
yogurt –> whole milk	0.40

表 6: Apriori 算法在 Groceries 数据集下发现的频繁项集（按支持度排序）

频繁项集	支持度	频繁项集	支持度
napkins	0.05	canned beer	0.08
beef	0.05	newspapers	0.08
curd	0.05	bottled beer	0.08
butter	0.06	citrus fruit	0.08
yogurt, whole milk	0.06	pastry	0.09
rolls/buns, whole milk	0.06	sausage	0.09
pork	0.06	shopping bags	0.10
coffee	0.06	tropical fruit	0.10
margarine	0.06	root vegetables	0.11
frankfurter	0.06	bottled water	0.11
domestic eggs	0.06	yogurt	0.14
brown bread	0.06	soda	0.17
whipped/sour cream	0.07	rolls/buns	0.18
fruit/vegetable juice	0.07	other vegetables	0.19
other vegetables, whole milk	0.07	whole milk	0.26
pip fruit	0.08		

结果表明，频繁项集集中在大小为 1 或 2 的集合中。为进一步探索 Groceries 数据集的关联规则，我在不同的支持度和置信度下使用 Apriori 算法对其进行关联规则分析，并将结果可视化，可参见图 3。



(a) 频繁项集个数　　(b) 关联规则个数

图 3: 频繁项集个数及关联规则个数与支持度和置信度关系

结果表明，在支持度大于 0.1 时，Groceries 数据集中基本已找不出满足条件的频繁项集；而置信度大于 0.5 时，几乎无法在剩余的频繁项集中发现关联规则。这个结果进一步证明了实验前的假设：每种商品出现的次数占总交易记录的比例偏低。同时，从置信度和关联规则个数的关系分析，Groceries 数据集中的交易记录存在一定的关联规则，但相应的置信度普遍偏低（0.5 以下）。

4 结论

本次作业中，我利用 Python 语言实现了 Apriori 算法，并通过在模拟交易数据集上与蛮力算法的结果比较，证明了算法实现的正确性。之后我使用 Apriori 算法对模拟交易数据集以及 Groceries 数据集进行了频繁项集挖掘，进而分析关联规则。

结果表明，Apriori 算法是迅速且有效的挖掘频繁项集的算法，在中小规模的数据集上有着良好的表现。限于计算资源，我没有进行大规模数据上的实验，但从理论上分析，Apriori 算法在大规模数据上的应用仍存在多个优化点，如对数据集的扫描次数，对候选项集支持度的计数方法等。

A 附录

A.1 模拟交易数据集的详细信息

我将模拟交易数据集的交易记录按照交易商品数分为 4 类，即 2、3、4、5 件。不同的商品件数按照不同的比例随机混合三类商品，具体混合规则可参见表 7。

表 7: 模拟交易数据集生成交易记录的混合规则

商品数	混合规则（括号中数字代表各类商品在记录中所占数量）
2	(2); (1, 1)
3	(3); (2, 1)
4	(4); (3, 1); (2, 1, 1)
5	(5); (4, 1); (3, 2); (2, 2, 1)

A.2 Apriori 算法实现代码

```
from itertools import chain, combinations
from collections import defaultdict

class Apriori(object):
    def __init__(self, f_name, sup=0.1, con=0.1):
        self.data = self._read_csv(f_name)
        self.sup = sup
        self.con = con
        self.items = []
        self.rules = []

    def _read_csv(self, f_name):
        with open(f_name, 'r') as f:
            for line in f:
                line = line.strip().rstrip(',')
                item = frozenset(line.split(','))
                yield item

    def run(self):
        # 小工具函数
        def _get_support(item):
            return float(freq_set[item]) / len(deals)
        def _get_subsets(item):
            return chain(*[combinations(item, i + 1) for i, a in
                enumerate(item)])
        def _join_set(item_set, id):
            return set([i.union(j) for i in item_set for j in item_set
                if len(i.union(j)) == id])

        # 初始化 L1 项集和交易数据
```

```
29          item_set, deals = self._init_data()
30
31          freq_set = defaultdict(int)
32          large_set = dict()
33
34          init_set = self._remove_item_set(item_set, deals, freq_set)
35          l_set = init_set
36
37          k = 2
38          while (l_set != set([])):
39              large_set[k - 1] = l_set
40              l_set = _join_set(l_set, k)
41              c_set = self._remove_item_set(l_set, deals, freq_set)
42              l_set = c_set
43              k += 1
44
45          # 组合结果
46          for key, value in large_set.items():
47              self.items.extend([(tuple(item), _get_support(item)) for
                    item in value])
48
49          for key, value in list(large_set.items())[1:]:
50              for item in value:
51                  _subsets = map(frozenset, [x for x in _get_subsets(
                        item)])
52                  for element in _subsets:
53                      remain = item.difference(element)
54                      if len(remain) > 0:
55                          con = _get_support(item) / _get_support(
                                element)
56                          if con >= self.con:
57                              self.rules.append(((tuple(element), tuple(
                                    remain)), con))
58
59      def _init_data(self):
60          deal_list = list()
61          item_set = set()
62
63          for d in self.data:
64              deal = frozenset(d)
65              deal_list.append(deal)
66
67              # 产生 L1 项集
68              for item in deal:
69                  item_set.add(frozenset([item]))
70
71          return item_set, deal_list
72
```

```
73      # 根据支持度阈值删除项集
74      def _remove_item_set(self, item_set, deals, freq_set):
75          res = set()
76          local_set = defaultdict(int)
77
78          # 统计
79          for item in item_set:
80              for d in deals:
81                  if item.issubset(d):
82                      freq_set[item] += 1
83                      local_set[item] += 1
84
85          # 删除
86          for item, count in local_set.items():
87              support = float(count) / len(deals)
88              if support >= self.sup:
89                  res.add(item)
90
91          return res
92
93      # 输出结果
94      def show(self):
95          print(u'———频繁项集———')
96          for item, sup in sorted(self.items, key=lambda items: items
                [1]):
97              print ('%s , %.2f' % (str(item), sup))
98          print (u'\n———关联规则———')
99          for rule, con in sorted(self.rules, key=lambda rules: rules
                [1]):
100             pre, post = rule
101             print ('%s --> %s , %.2f' % (str(pre), str(post), con))
102
103 if __name__ == "__main__":
104     a = Apriori('groceries.csv', 0.05, 0.2)
105     a.run()
106     a.show()
```